5 CONCLUSIONS AND FUTURE WORK

You're bound to be unhappy if you optimize everything.

— Donald Knuth

THE growing adoption of WebAssembly requires efficent code analysis and hardening techniques. This thesis contributes to this effort with a comprehensive set of methods and tools for Software Diversification in WebAssembly. It includes the technical contributions of this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

5.1 Summary of technical contributions

This thesis expands the field of Software Diversification within WebAssembly by introducing two distinct methods: compiler-based and binary-based approaches. Taking source code or LLVM bitcode as input, the compiler-based method generates WebAssembly variants. It uses enumerative synthesis and SMT solvers to produce numerous functionally equivalent variants. Importantly, these generated variants can be converted into multivariant binaries, thus enabling execution path randomization. Our compiler-based approach specializes in producing high-preservation variants, indicating that considerable effort is necessary to revert these variants to their initial program. However, a bottleneck in the variant generation process, specifically the use of SMT solvers for functional verification, undermines it when compared with the binary-based method. Furthermore, this method can only modify the code and function sections of WebAssembly binaries.

[0] Compilation probe time 2023/11/08 17:50:48

On the other hand, the method based on binary utilizes random e-graph traversals to create variants. This approach eliminates the need for modifications to existing compilers, ensuring compatibility with all existing WebAssembly binaries. Additionally, it offers a swift, efficient and novel method for generating variants through inexpensive random e-graph traversals. Consequently, our binary-based approach can produce variants at a scale at least one order of magnitude larger than our compiler-based approach. The binary-based method can generate variants by transforming any segment of the Wasm binary. However, the preservation of the generated variants is lower than the compiler-based approach.

We have developed three publicly accessible, open-source tools: CROW, MEWE, and WASM-MUTATE. CROW and MEWE follow a compiler-based approach, while WASM-MUTATE employs a binary-based method. These tools automate the diversification process, thereby enhancing practicality for deployment. Currently, WASM-MUTATE is improving WebAssembly compilers[1] in real-world contexts. Our tools serve as complements to each other, providing combined utility. For example, when the source code for a WebAssembly binary is inaccessible, WASM-MUTATE provides an efficient solution for generating code variants. On the other hand, for scenarios that require high preservation of variants, CROW and MEWE are particularly suitable. Last but not least, one can employ CROW and MEWE to generate a set of variants, which subsequently serve as rewriting rules for WASM-MUTATE. Furthermore, when practitioners require swift generation of variants, they could utilize WASM-MUTATE, accepting a decrease in preservation of the variants.

We have developed three open-source tools that are publicly accessible: CROW, MEWE, and WASM-MUTATE. CROW and MEWE utilize a compiler-based approach, whereas WASM-MUTATE employs a method based on binary. These tools automate the process of diversification, thereby increasing their practicality for deployment. At present, WASM-MUTATE is enhancing WebAssembly compilers[2] in real-world situations. Our tools are complementary, providing combined utility. For instance, when the source code for a WebAssembly binary is unavailable, WASM-MUTATE offers an efficient solution for the generation of code variants. On the other hand, CROW and MEWE are particularly suited for scenarios that require a high level of variant preservation. Finally, one can use CROW and MEWE to generate a set of variants, which can then serve as rewriting rules for WASM-MUTATE. Moreover, when practitioners need to quickly generate variants, they could employ WASM-MUTATE, despite a potential decrease in the preservation of variants.

[1] `https://github.com/bytecodealliance/wasm-tools`
[2] `https://github.com/bytecodealliance/wasm-tools`

5.2 Summary of empirical findings

We demonstrate the practical application of Offensive Software Diversification in WebAssembly. In particular, we diversify 33 WebAssembly cryptomalwares automatically, generating numerous variants in just a few minutes. These variants successfully evade detection by state-of-the-art malware detection systems. Our research confirms the existence of opportunities for the malware detection community to strengthen the automatic detection of cryptojacking WebAssembly malware. Specifically, developers can improve the detection of WebAssembly malware by using multiple malware oracles, or meta-oracles. Additionally, these practitioners could employ feedback-guided diversification to identify specific transformations their implementation is susceptible to. For instance, our study found that the addition of arbitrary custom sections to WebAssembly binaries is a highly effective transformation for evading detection. In practice, no WebAssembly engine uses custom sections, so injecting them does not impact the performance of the WebAssembly binary. Developers could enhance their detection systems' robustness by normalizing the WebAssembly binaries, removing custom sections. This logic also applies to other transformations, such as adding unreachable code, another effective method for evading detection.

On the other side of the coin, our techniques enhance overall security from a Defensive Software Diversification perspective. We facilitate the deployment of unique, diversified and hardened WebAssembly binaries. As previously demonstrated, WebAssembly variants produced by our tools exhibit improved resistance to side-channel attacks. Our tools generate variants by modifying malicious code patterns such as embedded timers used to conduct timing side-channel attacks. Simultaneously, they can produce variants that introduce noise into the execution side-channels of the original program, concurrently altering the memory layout of the JITed code generated by the host engine.

Our methods remarkably generate thousands of variants in mere minutes. The swift production of these variants is due to the rapid transformation of WebAssembly binaries. This swift generation of variants is particularly advantageous in highly dynamic scenarios such as FaaS and CDN platforms. We have empirically tested the effectiveness of moving target defense techniques[**?**] on the Fastly edge computing platform. In this scenario, we incorporate multivariant executions[**?**]. Fastly can redeploy a WebAssembly binary across its 73 datacenters worldwide in 13 seconds on average. This enables the practical deployment of a unique variant per node using our tools. However, a 13-second window may still pose a risk despite each node potentially hosting a distinct WebAssembly variant. To mitigate this, we use multivariant binaries, invoking a unique variant with each execution. Our tools can generate dozens of unique variants every few seconds, each serving as a multivariant binary packaging thousands of other variants. This illustrates the real-world application of Defensive Software Diversification to a WebAssembly standalone scenario.

5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline two concrete directions.

Program Normalization: Malware detection is a well-known and challenging issue [**?**]. The problem intensifies when considering malware in the context of software monoculture, especially when the malware is assumed to be identically replicated across victims. We have successfully used Software Diversification to evade malware detection. This shows that, if applicable, the reality of malware monoculture can be exploited by attackers using Software Diversification for offensive purposes. In previous discussions, we highlighted the use of feedback-guided diversification in identifying specific transformations to which an implementation is susceptible. However, we have found another approach to mitigate malware evasion. Specifically, instead of generating variants to evade detection, we propose to canonize variants to a primer program.

TODO One paragraph on SOTA

Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be JITed to machine code, effectively eliminating malware variants. This approach could substantially enhance the efficiency and precision of malware detection systems.

TODO One paragraph about how I would do it.

Meta-oracles: Our experiment results in Section 4.1 indicate that VirusTotal surpasses MINOS in detecting WebAssembly cryptojacking. The primary factor contributing to this is VirusTotal's utilization of a broader range of antivirus vendors, which employs various detection strategies. On the other hand, MINOS functions as a binary oracle. This evidence supports the use of multiple malware oracles (meta-oracles) in identifying cryptojacking malware in browsers. In the context of WebAssembly, given the existence of numerous and diverse Wasm-specific detection mechanisms, this strategy is both practical and feasible, yet not explored in the literature.

TODO Motivation. Why this is important?

TODO One paragraph on SOTA

TODO One paragraph about how I would do it.